3

Symmetric Cryptography

In this chapter, you will be introduced to the concepts, theory, and practical aspects of **symmetric cryptography**. More focus will be given to the elements that are specifically relevant in the context of blockchain technology. This chapter will provide the concepts required to understand the material covered in later chapters.

You will also be introduced to applications of cryptographic algorithms so that you can gain hands-on experience in the practical implementation of cryptographic functions. For this, the OpenSSL command-line tool is used.

This chapter will cover the following topics:

- Introducing cryptography
- Cryptographic primitives
- Advanced Encryption Standard

For the exercises in this chapter, the installation of OpenSSL is necessary. On the Ubuntu Linux distribution, OpenSSL is usually already available. On macOS, LibreSSL is already available. However, it can be installed using the following command:

```
$ sudo apt-get install openssl
```

Examples in this chapter have been developed using OpenSSL 3.0.1. You are encouraged to use this specific version, as all examples in the chapter have been developed and tested with it. If you are running a version other than version 3, the examples may still work but that is not guaranteed.

Introducing cryptography

Cryptography is the science of making information secure in the presence of adversaries. **Ciphers** are algorithms used to encrypt or decrypt data so that if intercepted by an adversary, the data is meaningless to them without **decryption**, which requires a secret key.

Cryptography is primarily used to provide a confidentiality service. On its own, it cannot be considered a complete solution; rather, it serves as a crucial building block within a more extensive security system to address a security problem. For example, securing a blockchain ecosystem requires many different cryptographic primitives, such as hash functions, symmetric key cryptography, digital signatures, and public key cryptography, which we will discuss in the next chapter.

In addition to a confidentiality service, cryptography also provides other security services such as integrity, authentication (entity authentication and data origin authentication), and non-repudiation. Additionally, accountability is provided, which is a requirement in many security systems.

A generic cryptographic model is shown in the following diagram:

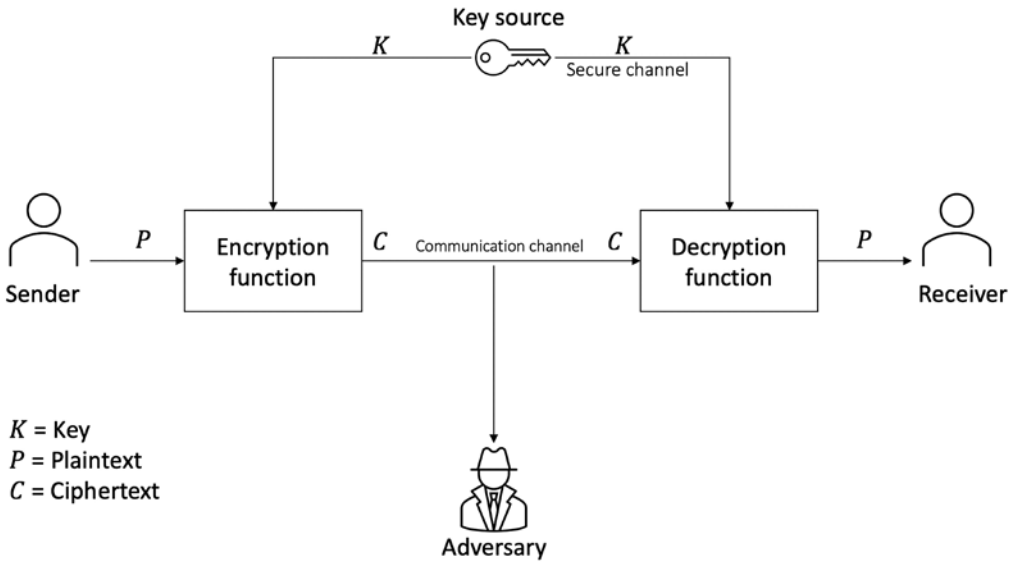


Figure 3.1: A generic encryption and decryption model

In the preceding diagram, *P*, *C*, and *K* represent plaintext, ciphertext, and key (data that is used to encrypt plaintext and decrypt ciphertext) respectively.

Services provided by cryptography

We mentioned the fact that cryptography provides various services. In the following section, we'll introduce these services.

Confidentiality is the assurance that information is only available to authorized entities.

Integrity is the assurance that information is modifiable only by authorized entities.

Authentication provides assurance about the identity of an entity or the validity of a message. There are two types of authentication mechanisms, namely, entity authentication and data origin authentication, which are discussed in the following sections.

Entity authentication is the assurance that an entity is currently involved and active in a communication session. Traditionally, users are issued a username and password that is used to gain access to the various platforms with which they are working. This practice is known as **single-factor authentication**, as there is only one factor involved, namely, something you know—that is, the password and username.

This type of authentication is not very secure for a variety of reasons, for example, password leakage; therefore, additional factors are now commonly used to provide better security. The use of additional techniques for user identification is known as **multi-factor authentication**:

- The first method uses *something you have*, such as a hardware token or a smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. A user who has access to the hardware token and knows the login credentials will be able to access the system. If the hardware token is inaccessible (for example, lost or stolen) or the login password is forgotten by the user, access to the system will be denied. The hardware token won't be of any use on its own unless the login password (*something you know*) is also known and used in conjunction with the hardware token.
- The second method uses *something you are*, which uses biometric features to identify the user. With this method, a user's fingerprint, retina, iris, or hand geometry is used to ensure that the user was indeed present during the authentication process. However, careful implementation is required to guarantee a high level of security, as some research has suggested that biometric systems can be circumvented under specific conditions.

Data origin authentication is another type of authentication, which is also known as **message authentication**. It is an assurance that the source of the information is indeed verified. Data origin authentication guarantees data integrity because, if a source is corroborated, then the data must not have been altered. Various methods are used for this type of authentication, such as **message authentication codes** (**MACs**) and digital signatures, which we'll cover later in this chapter and the next chapter.

Another important assurance provided by cryptography is **non-repudiation**. It is the assurance that an entity cannot deny a previous commitment or action by providing incontrovertible cryptographic evidence. This property is essential in debatable situations whereby an entity has denied the actions performed, for example, the placement of an order on an e-commerce system. Non-repudiation has been an active research area for many years. Disputes in electronic transactions are a common issue, and there is a need to address them to increase consumers' confidence levels in such services.

The non-repudiation protocol usually runs in a communications network, and it is used to provide evidence that an action has been taken by an entity (originator or recipient) on the network. In this context, two communications models can be used to transfer messages from originator *A* to recipient *B*:

- A message is sent directly from originator *A* to recipient *B*
- A message is sent to a delivery agent from originator *A*, which then delivers the message to recipient *B*

The primary requirements of a non-repudiation protocol are fairness, effectiveness, and timeliness. In many scenarios, there are multiple participants involved in a transaction. For example, in electronic trading systems, there can be clearing agents, brokers, traders, and other entities who can be involved in a single transaction. To address this problem, **multi-party non-repudiation** (**MPNR**) protocols have been developed.

Accountability is the assurance that actions affecting security can be traced back to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are vital to trace an entity's actions, such as when a trade is placed in an audit record with the date and timestamp and the entity's identity is generated and saved in the log file. This log file can optionally be encrypted and be part of the database or a standalone ASCII text log file on a system.

To provide all of the services discussed in this section, different cryptographic primitives are used, which are presented in the next section.

Cryptographic primitives

Cryptographic primitives are the basic building blocks of a security protocol or system. A **security protocol** is a set of steps taken to achieve the required security goals by utilizing appropriate security mechanisms. Various types of security protocols are in use, such as authentication protocols, non-repudiation protocols, and key management protocols.

The taxonomy of cryptographic primitives can be visualized as shown here:

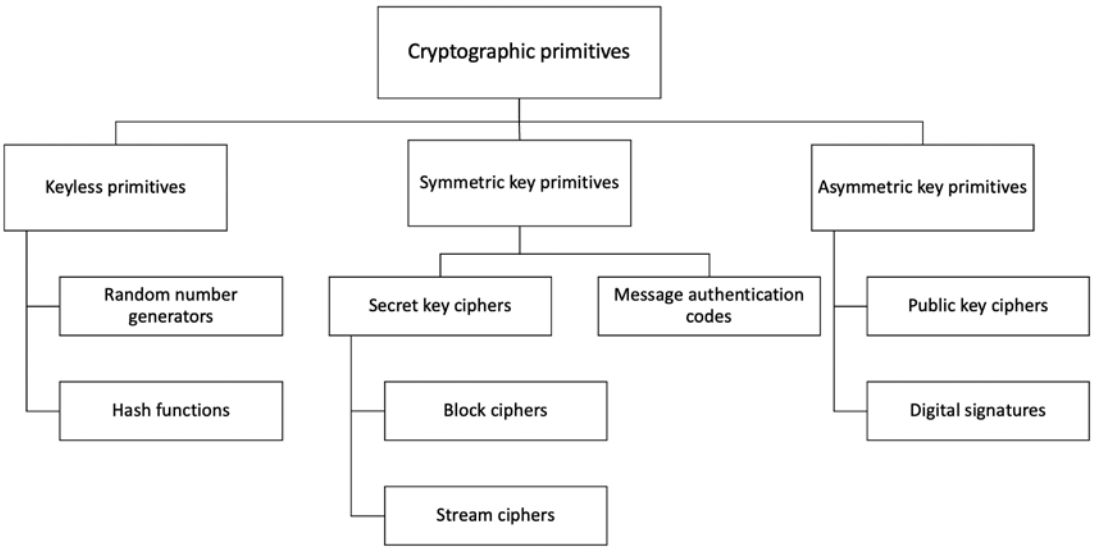


Figure 3.2: Cryptographic primitives

As shown in the preceding cryptographic primitive taxonomy diagram, cryptography is mainly divided into three categories: **keyless primitives, symmetric key primitives, and asymmetric key primitives**.

Keyless primitives and symmetric cryptography are discussed further in the next section, and we will cover asymmetric cryptography, or public key cryptography, in *Chapter 4*, *Asymmetric Cryptography*.

Keyless primitives

In this section, we will introduce two keyless primitives, namely, random numbers and hash functions.

Random numbers

Randomness provides an indispensable element for the security of cryptographic protocols. It is used for the generation of keys and in encryption algorithms. Randomness ensures that the operations of a cryptographic algorithm do not become predictable enough to allow cryptanalysts to infer the outputs and operations of the algorithm, which will make the algorithm insecure. It is quite a feat to generate suitable randomness with a high degree of uncertainty, but there are methods that ensure an adequate level of randomness is generated for use in cryptographic algorithms.

There are two categories of the source of randomness, namely, **random number generators** and **pseudorandom number generators**.

Random number generators

Random number generators (**RNGs**) are software or hardware systems that make use of the randomness available in the real world, called **real randomness**. This can be temperature variations, thermal noises from various electronic components, or acoustic noise. Other sources are based on physical phenomena such as keystrokes, mouse cursor movements, or disk movements of a running computer system. These types of sources of randomness are not very practical due to the difficulty of acquiring this data or not having enough entropy. Also, these sources are not always available or could be available only for a limited time.

Pseudorandom number generators

Pseudorandom number generators (**PRNGs**) are deterministic functions that work on the principle of using a random initial value called a **seed** to produce a random-looking set of elements. PRNGs are commonly used to generate keys for encryption algorithms. A common example of a PRNG is **Blum-Blum-Shub** (**BBS**). PRNGs are a better alternative to RNGs due to their reliability and deterministic nature.

More information on BBS is available in the following original research paper, *Blum, L., Blum, M., and Shub, M., 1986. A simple unpredictable pseudo-random number generator. SIAM Journal on computing, 15(2), pp.364–383*:

`https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf`

Generating random strings

The following command can be used to generate a pseudorandom string using OpenSSL:

```
$ openssl rand -hex 16
```

It will produce the output random 16-byte string encoded in hex.

```
06532852b5da8a5616dfade354a9f270
```

There are other variations that you can explore more with the OpenSSL command-line tool. Further information and help can be retrieved with the following command:

```
$ openssl help
```

In the next section, we will look at hash functions, which play a crucial role in the development of blockchain.

Hash functions

Hash functions are used to create fixed-length digests of arbitrarily long input strings. Hash functions are **keyless**, and they provide a **data integrity service**. They are usually built using iterated and dedicated hash function construction techniques.

Various families of hash functions are available, such as MD, SHA1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used for digital signatures and MACs, such as HMACs.

Hash functions are also typically used to provide data integrity services. These can be used both as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications use hash functions as a means of generating PRNGs. There are two practical properties of hash functions that must be met depending on the level of integrity required:

- **Compression of arbitrary messages into fixed-length digests**: This property relates to the fact that a hash function must be able to take an input text of any length and output a fixed-length compressed message. Hash functions produce a compressed output in various bit sizes, usually between 128-bit and 512-bit.
- **Easy to compute**: Hash functions are efficient and fast one-way functions. It is required that hash functions be very quick to compute regardless of the message size. The efficiency may decrease if the message is too big, but the function should still be fast enough for practical use.

There are also three security properties that must be met, depending on the level of integrity:

- **Pre-image resistance**: This property states that if given a value y, it is computationally infeasible (almost impossible) to find a value x such that $h(x)= y$. Here, h is the hash function, x is the input, and y is the hash. The first security property requires that y cannot be reverse computed to x. x is considered a pre-image of y, hence the name **pre-image resistance**. This is also called a one-way property.
- **Second pre-image resistance**: The **second pre-image resistance** property states that given x it is computationally infeasible to find another value x' such that $x' \neq x$ and $h(x') = h(x)$. This property is also known as **weak collision resistance**.
- **Collision resistance**: The **collision resistance** property states that it is computationally infeasible to find two distinct values x' and x such that $h(x') = h(x)$. In other words, two different input messages should not hash to the same output. This property is also known as strong collision resistance.

These security properties are depicted in the following diagram:

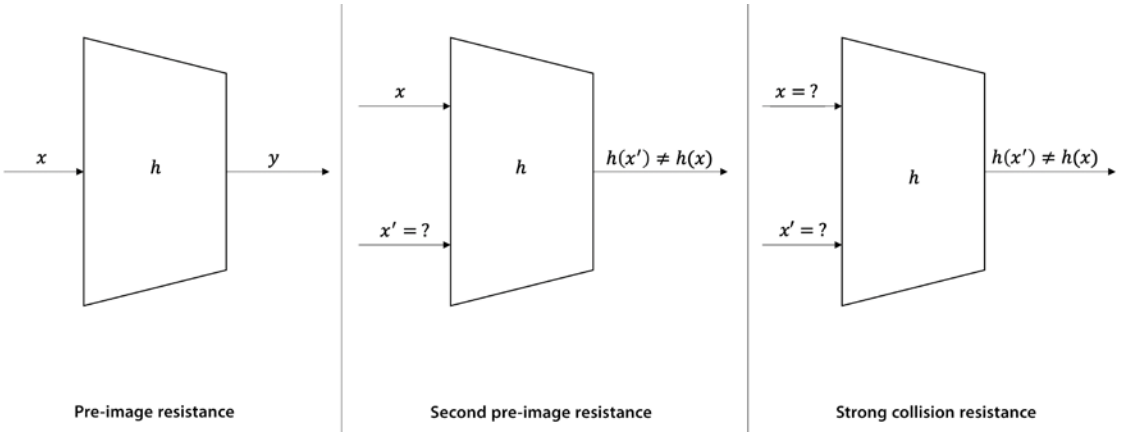


Figure 3.3: Three security properties of hash functions

Due to their very nature, hash functions will always have some collisions. This is a situation where two different messages hash to the same output. However, they should be computationally impractical to find. A concept known as the **avalanche effect** is desirable in all cryptographic hash functions. The avalanche effect specifies that a small change, even a single character change in the input text, will result in an entirely different hash output.

Hash functions are usually designed by following an iterated hash functions approach. With this method, the input message is compressed in multiple rounds on a block-by-block basis to produce the compressed output. A popular type of iterated hash function is the **Merkle-Damgard construction**. This construction is based on the idea of dividing the input data into equal block sizes and then feeding them through the compression functions in an iterative manner. The collision resistance of the property of compression functions ensures that the hash output is also collision-resistant. In addition to Merkle-Damgard, there are various other constructions of compression functions proposed by researchers, for example, Miyaguchi-Preneel and Davies-Meyer.

Multiple categories of hash functions are introduced in the following section.

Message digest functions

Message digest (**MD**) functions were prevalent in the early 1990s. MD4 and MD5 fall into this category. Both MD functions were found to be insecure due to message collisions found and are not recommended for use anymore. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms

The following list describes the most common **secure hash algorithms** (**SHAs**):

- **SHA-0**: This is a 160-bit function introduced by the U.S. **National Institute of Standards and Technology** (**NIST**) in 1993.

- **SHA-1**: SHA-1 was introduced in 1995 by NIST as a replacement for SHA-0. This is also a 160-bit hash function. SHA-1 is used commonly in SSL and TLS implementations. It should be noted that SHA-1 is now considered insecure, and it is being deprecated by certificate authorities. Its usage is discouraged in any new implementations.
- **SHA-2**: This category includes four functions defined by the number of bits of the hash: SHA-224, SHA-256, SHA-384, and SHA-512.
- **SHA-3**: This is the latest family of SHA functions. SHA3-224, SHA3-256, SHA3-384, and SHA3-512 are members of this family. SHA-3 is a NIST-standardized version of Keccak.
- **RIPEMD**: RIPEMD is the acronym for **RACE Integrity Primitives Evaluation Message Digest**. It is based on the design ideas used to build MD4. There are multiple versions of RIPEMD, including 128-bit, 160-bit, 256-bit, and 320-bit.
- **Whirlpool**: This is based on a modified version of the Rijndael cipher known as *W*. It uses the Miyaguchi-Preneel compression function, which is a type of one-way function used for the compression of two fixed-length inputs into a single fixed-length output. It is a single-block length compression function.

Hash functions have many practical applications ranging from simple file integrity checks and password storage to use in cryptographic protocols and algorithms. They are used in **Peer to Peer** (**P2P**) networks, P2P file sharing, virus fingerprinting, bloom filters, Merkle trees, Patricia trees, and distributed hash tables.

In the following section, you will be introduced to the design of SHA-256 and SHA-3. Both of these are used in Bitcoin and Ethereum, respectively. Ethereum uses Keccak, which is the original algorithm presented to NIST, rather than NIST standard SHA-3. NIST, after some modifications, such as an increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

SHA-256

SHA-256 has an input message size limit of 2^{64} - 1 bits. The block size is 512 bits, and it has a word size of 32 bits. The output is a 256-bit digest. The compression function processes a 512-bit message block and a 256-bit intermediate hash value. There are two main components of this function: the compression function and a message schedule. The algorithm works as follows, in nine steps:

Pre-processing:

a. Padding of the message is used to adjust the length of a block to 512 bits if it is smaller than the required block size of 512 bits.
b. Parsing the message into message blocks, which ensures that the message and its padding are divided into equal blocks of 512 bits.

c. Setting up the initial hash value, which consists of the eight 32-bit words obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. These initial values are fixed and chosen to initialize the process. They provide a level of confidence that no backdoor exists in the algorithm.

Hash computation:

a. Each message block is then processed in a sequence, and it requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
b. The message schedule is prepared.
c. Eight working variables are initialized.
d. The compression function runs 64 times.
e. The intermediate hash value is calculated.
f. Finally, after repeating steps 5 through 8 until all blocks (chunks of data) in the input message are processed, the output hash is produced by concatenating intermediate hash values.

At a high level, SHA-256 can be visualized in the following diagram:

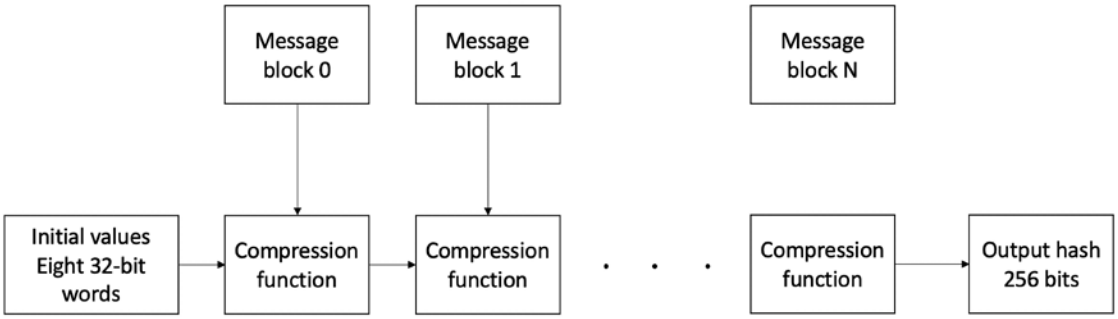


Figure 3.4: SHA-256 high-level overview

As shown in the preceding diagram, SHA-256 is a Merkle-Damgard construction that takes the input message and divides it into equal blocks (chunks of data) of 512 bits. Initial values (or initial hash values) or the initialization vector are composed of eight 32-bit constant words (*a, b, c, d, e, f, g* —256 bits each) that are fed into the compression function with the first message block. Subsequent blocks are fed into the compression function until all blocks are processed, and finally, the output hash is produced.

The compression function of SHA-256 is shown in the following diagram:

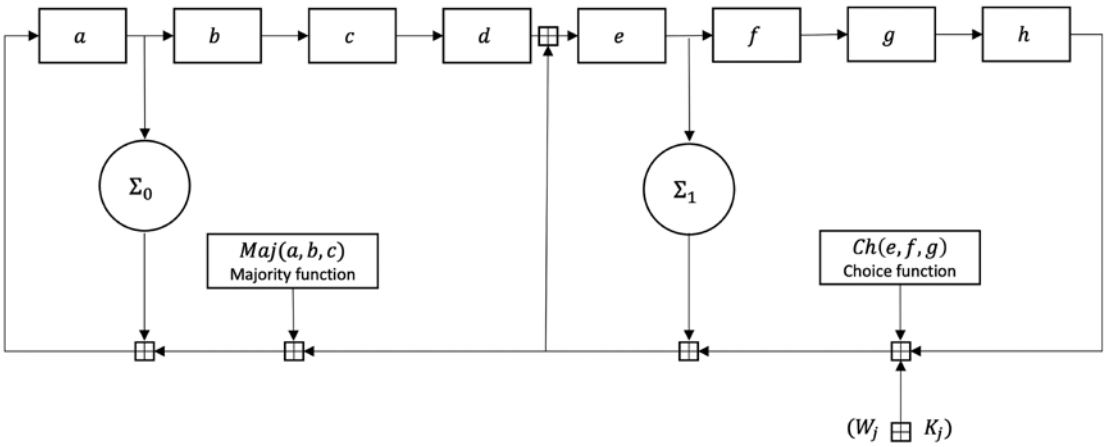


Figure 3.5: SHA-2 compression function

In the preceding diagram, *a, b, c, d, e, f, g* and *h* are the registers for eight initial pre-determined constants and then for intermediate hash values for the next blocks. *Maj* and *Ch* functions are defined as the formulae shown below:

$$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

$$Ch(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$$

where $\wedge$ is bitwise AND, $\oplus$ is bitwise XOR, and $\neg$ is bitwise NOT. XOR can be replaced with bitwise OR without any change in the output. The functions operate on vectors of 32 bits.

Maj is the "majority" function where the output produced is based on the majority of the inputs. In other words, if most of the inputs are 1 then the output is 1; otherwise, 0. Here 3 input bits x, y and z produce the output.

Ch is the choice function where a choice is made between the inputs depending upon the value of one of the inputs. It is like an "if .. then .. else" construct in programming, or a data selector, or input selector. It works on the following principle:

$$Ch(e,f,g) = \begin{cases} f, & if\ e = 1 \\ g, & if\ e = 0 \end{cases}$$

Here *e*, *f*, *g* are inputs, and depending on whether $e = 1$ or $e = 0$, either *f* or *g* is selected.

As shown in *Figure 3.5*, *Maj* and *Ch* functions are applied bitwise. Σ_0 and Σ_1 perform bitwise rotation. $\Sigma_0(a)$ returns $(a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)$, and $\Sigma_1(e)$ returns $(e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)$. $\boxplus$ means addition modulo 2^{32} for SHA-256.

The mixing constants are W_i and K_j, which are added in the main loop (compressor function) of the hash function, which runs 64 times.

With this, our introduction to SHA-256 is complete, and next we explore a newer class of hash functions known as the SHA-3 algorithm.

SHA-3 (Keccak)

The structure of **SHA-3** is very different from that of SHA-1 and SHA-2. The key idea behind SHA-3 is based on unkeyed permutations, as opposed to other typical hash function constructions that used keyed permutations. **Keccak** also does not make use of the Merkle-Damgard transformation that is commonly used to handle arbitrary-length input messages in hash functions. A newer approach, called **sponge and squeeze construction**, is used in Keccak. It is a random permutation model. Different variants of SHA-3 have been standardized, such as SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. SHAKE128 and SHAKE256 are **extendable-output functions** (**XOFs**), which allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model, which is the basis of SHA-3 or Keccak. Analogous to a sponge, the data (*m* input data) is first absorbed into the sponge after applying padding. It is then changed into a subset of the permutation state using **XOR** (**exclusive OR**), and then the output is squeezed out of the sponge function that represents the transformed state. The rate *r* is the input block size of the sponge function, while capacity *c* determines the security level:

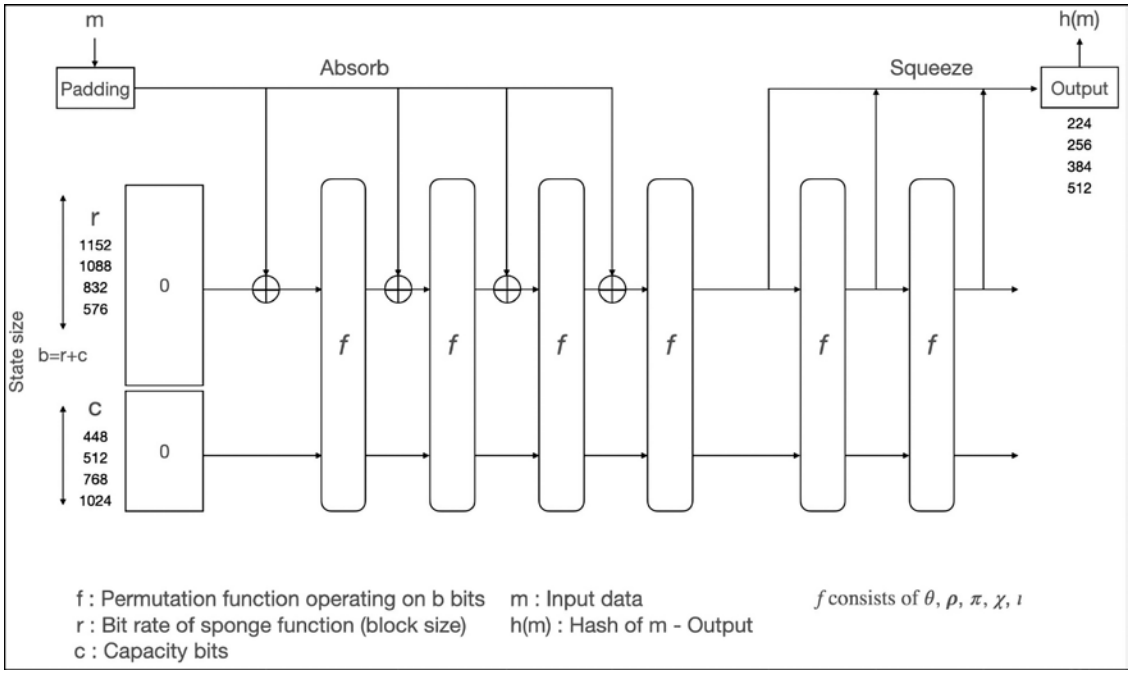


Figure 3.6: The SHA-3 absorbing and squeezing function in SHA3

In the preceding diagram, state size *b* is calculated by adding bit rate *r* and capacity bits *c*. *r* and *c* can be any values as long as sizes of *r* + *c* are 25, 50, 100, 200, 400, 800, or 1,600. The state is a 3-dimensional bit matrix. The initial state is set to 0.

The data *m* is entered into the absorb phase block by block via XOR ⊕ after applying padding.

The following table shows the value of bit rate *r* (block size) and capacity *c* required to achieve the desired output hash size under the most efficient setting of *r* + *c* = 1600:

r (block size)	c (capacity)	Output hash size
1152	448	224
1088	512	256
832	768	384
576	1024	512

The function *f* is a permutation function. It contains five transformation operations:

- θ(Theta): XOR bits in the state, used for mixing
- ρ (Rho): Diffusion function performing rotation of bits
- π (Pi): Diffusion function
- χ (XOR): Each bit, bitwise combine
- ι (Iota): Combination with round constants

The details of each transformation operation are beyond the scope of this book. The key idea is to apply these transformations to achieve the **avalanche effect**, which we introduced earlier in this chapter. These five operations combined form a **round.** In the SHA-3 standard, the number of rounds is 24 to achieve the desired level of security.

We will see some applications and examples of constructions built using hash functions such as Merkle trees in *Chapter 9*, *Ethereum Architecture*.

Encrypting messages with SHA-256

The following command will produce a hash of 256 bits of `Hello` messages using the SHA-256 algorithm:

```
$ echo -n 'Hello' | openssl dgst -sha256
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

Now we run the following command to see the avalanche effect in action:

```
$ echo -n 'hello' | openssl dgst -sha256
(stdin)= 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Note that even a small change in the text, such as changing the case of the letter `H`, results in a big change in the output hash:

```
Hello:
18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:80:07:
d1:76:48:26:38:19:69
hello:
2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5e:73:
04:33:62:93:8b:98:24
```

Usually, hash functions do not use a key. Nevertheless, if they are used with a key, then they can be used to create **MACs**.

Applications of cryptographic hash functions

There are various constructs that have been built using basic cryptographic parameters to solve different problems in computing. These constructs are also used in blockchains to provide various protocol-specific services. For example, hash functions are used to build Merkle trees, which are used to efficiently and securely verify large amounts of data in distributed systems. Some other applications of hash functions in blockchains are to provide several security services.

These services are listed here:

- Hash functions are used in cryptographic puzzles such as the **proof of work** (**PoW**) mechanism in Bitcoin. Bitcoin's PoW makes use of the SHA-256 cryptographic hash function.
- The generation of addresses in blockchains. For example, in Ethereum, blockchain accounts are represented as addresses. These addresses are obtained by hashing the public key with the Keccak-256 hash algorithm and then using the last 20 bytes of this hashed value.
- Message digests in digital signatures.
- The creation of Merkle trees to guarantee the integrity of transaction structure in the blockchain. Specifically, this structure is used to quickly verify whether a transaction is included in a block or not. However, note that Merkle trees on their own are not a new idea; it has just been made more popular with the advent of blockchain technology.

Merkle trees are the core building blocks of all blockchains, for example, Bitcoin and Ethereum. We will explore Merkle trees in detail now.

Merkle tree

The Merkle tree was introduced by Ralph Merkle. **Merkle trees** enable the secure and efficient verification of large datasets. A diagram of a Merkle tree is shown below:

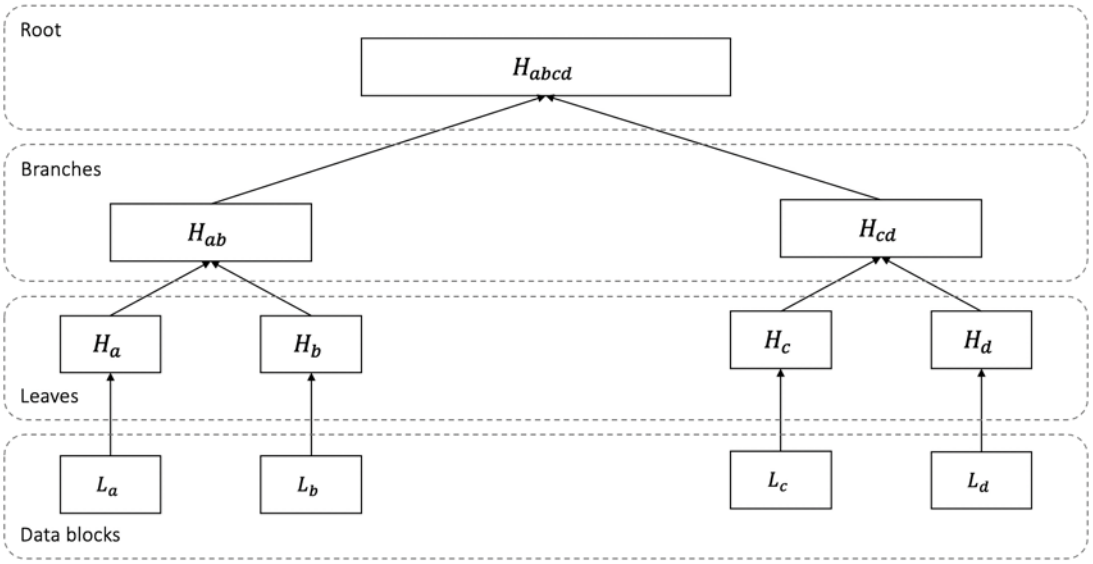


Figure 3.7: A Merkle tree

A Merkle tree is a binary tree in which the inputs are first placed at the leaves (nodes with no children), and then the values of pairs of child nodes are hashed together to produce a value for the parent node (internal node), until a single hash value known as a **Merkle root** is achieved. This structure helps to quickly verify the integrity of the entire tree (entire dataset), but just by verifying the Merkle root on top of the Merkle tree, because if any change occurs in any of the hashes in the tree, the Merkle root will also change.

A Merkle tree is used to prove that a data block B_i is a member of a set of N data blocks, formally $B \in \{B_1, .., B_N\}$. With a Merkle root and a candidate data block, using Merkle trees, we can prove that a data block exists within a set. This proof is called Merkle proof and involves obtaining a set of hashes called "Merkle path" for a given data block and Merkle root. In other words, the Merkle path for a data element (or block) is the minimum chain of hashes required to recreate the Merkle root by repeatedly hashing and concatenating until the Merkle root is created. For example, in *Figure 3.7*, if the existence of L_a needs to be proven, then the Merkle path would be $H_a \rightarrow H_{ab}$ up to the root H_{abcd}. We can prove the existence of data block L_a by recreating the Merkle root through this path and comparing it with the provided Merkle root; if the Merkle roots don't match, then we can say that the data element in question is not in the Merkle tree or vice versa. The Merkle path is also known as the authentication path.

This is the reason why the integrity of the system can be verified quickly by just looking at the Merkle root. Another advantage of Merkle trees is that there is no requirement to store large amounts of data, only the hashes of the data, which are fixed-length digests of the large dataset. Due to this property, the storage and management of Merkle trees are easy and efficient as they take up a very small amount of space for storage. Also, since the tree is storage efficient, the relevant proofs for integrity are also smaller in size and quick to transmit over the network, thus making them bandwidth-efficient over the network.

Merkle Patricia trie

To understand Patricia trees, we need to understand **trie** first. A trie, or a **digital tree**, is an ordered tree data structure used to store a dataset. The **Practical Algorithm to Retrieve Information Coded in Alphanumeric** (**PATRICIA**) tree, also known as a **Radix tree**, is a compact representation of a trie in which a node that is the only child of a parent is merged with its parent. The keys represent the path to reach a node. The nodes that share the same key can share the same path, thus making it an efficient way of finding common prefixes while utilizing a small amount of memory.

A **Merkle-Patricia tree** is a tree that has a root node that contains the hash value of the entire data structure. The Merkle-Patricia tree combines Merkle and Patricia trees where Patricia is used for efficient storage and Merkle enables tamper-proof data validation. Patricia tree is also modified to store hexadecimal strings instead of bits and support 16 branches. Merkle Patricia tree has four types of nodes:

- **Null** nodes, which are non-existent nodes represented as empty strings.
- **Branch** nodes, which are 17-item nodes used for branching. They have 16 possible branches from 0–F, and the 17th item (value) stores a value only if the node is terminating.
- **Extension** nodes, which are 2-item nodes containing a path and a value. They are used for compression, where a common part of multiple keys is stored, i.e., a shared nibble.
- **Leaf** nodes, which are 2-item node containing a path and a value. The leaf node is the terminating node in the path with no children and groups common key suffixes in the path as the compression technique.

Leaf nodes and extension nodes are distinguished by a hex prefix value. A leaf node with an even number of nibbles has a prefix of 2; for an odd number of nibbles, 3 is used. Extension nodes with an even number of nibbles have the prefix 0, and for an odd number the prefix is 1.

The figure below shows how a Merkle-Patricia tree is used to store keys and values (accounts and balances in Ethereum):

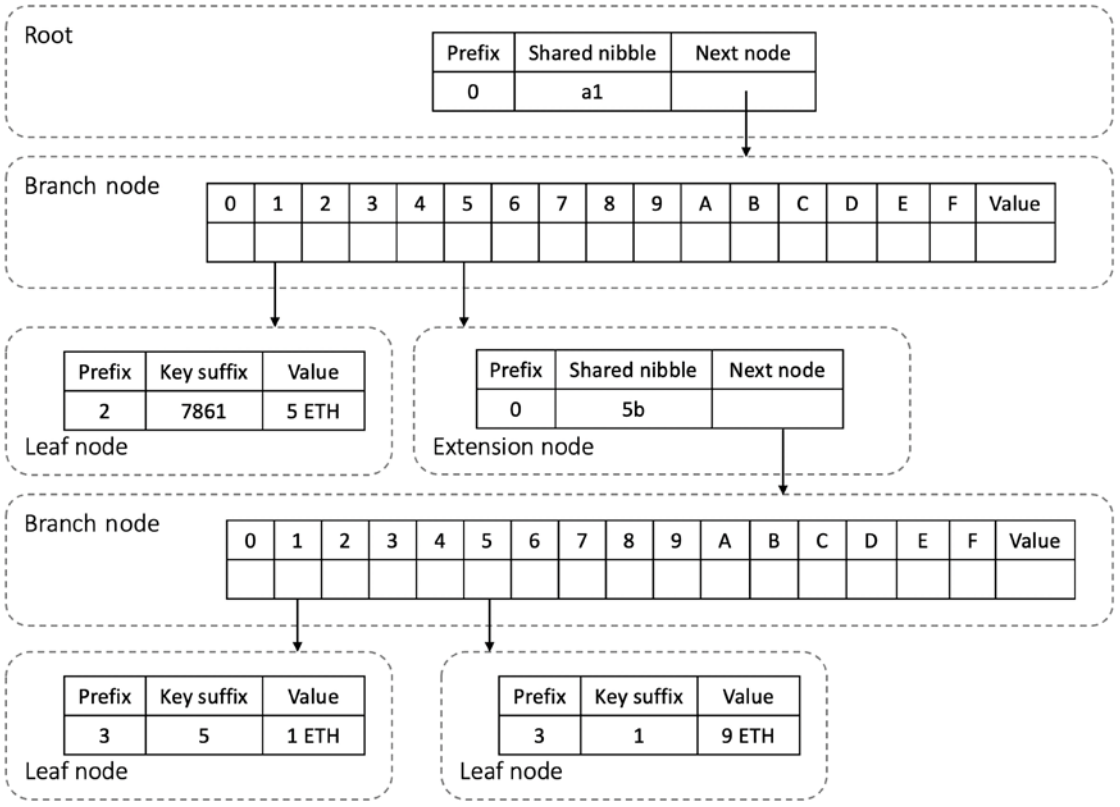


Figure 3.8: Merkle-Patricia tree

In the diagram above, key a117861 has a value of 5 ETH, key a155b15 has a value of 1 ETH, and key a155b51 has a value of 9 ETH. Notice that a1 is a common prefix (a shared nibble) and is grouped at the root extension node. Then a leaf node (with no child) with the key suffix 7861 branches out from the branch node. Further, another extension node, with the shared nibble (prefix) 5b branches out and leads to another branch node, finally ending with two leaf nodes with no further children with the key suffixes 5 and 1. Merkle-Patricia trees are used in the Ethereum blockchain to store key-value pairs where the root is hashed using SHA3 and included in the header of the block.

Distributed hash tables

A hash table is a data structure that is used to map keys to values. Internally, a hash function is used to calculate an index into an array of buckets from which the required value can be found. Buckets have records stored in them using a hash key and are organized into a particular order.

With the definition provided above in mind, we can think of a **distributed hash table** (**DHT**) as a data structure where data is spread across various nodes, and nodes are equivalent to buckets in a P2P network. The following diagram shows how a DHT works:

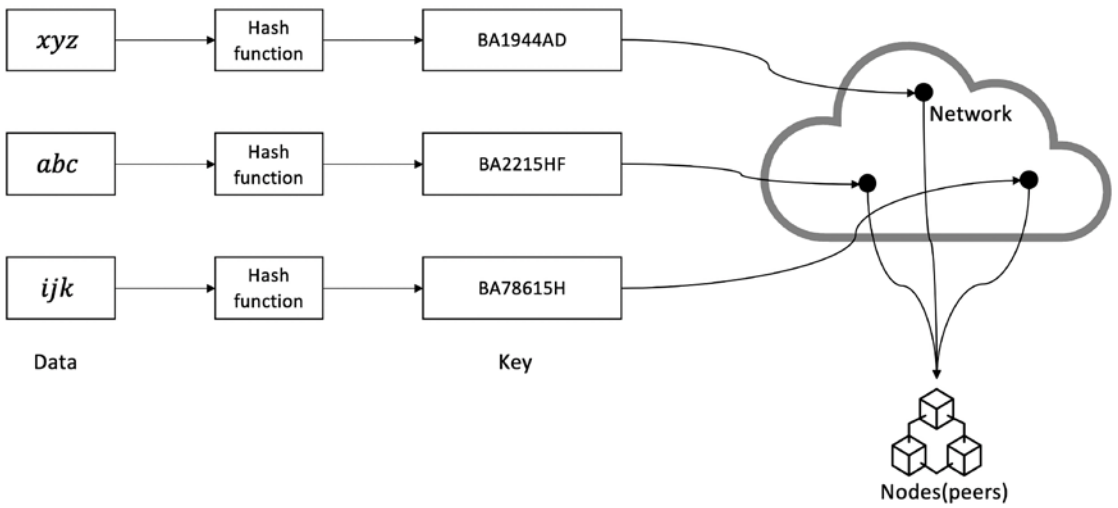


Figure 3.9: Distributed hash table

In the preceding diagram, data is passed through a hash function, which then generates a compact key. This key is then linked with the data (values) on the P2P network. When users on the network request the data (via the filename), the filename can be hashed again to produce the same key, and any node on the network can then be requested to find the corresponding data. A DHT provides decentralization, fault tolerance, and scalability.

In this section, we covered various applications of cryptographic hash functions. Hash functions are of particular importance in blockchains, as they are key to constructing Merkle trees, which are used in blockchains for the efficient and fast verification of large datasets. We will see how MACs and HMACs work after we've introduced symmetric key cryptography, because these constructions make use of keys for encryption.

Symmetric key primitives

Symmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is the same one that is used for decrypting the data. Thus, it is also known as **shared key cryptography**. The key must be established or agreed upon before the data exchange occurs between the communicating parties. This is the reason it is also called **secret key cryptography**.

Other types of keys are **public keys** and **private keys**, which are generated in pairs for public key cryptography or asymmetric cryptography. Public keys are used for encrypting the plaintext, whereas private keys are used for decryption and are expected to be kept secret by the receiver.

Keys can also be **ephemeral** (temporary) or **static**. Ephemeral keys are intended to be used only for a short period of time, such as in a single session between the participants, whereas static keys are intended for long-term usage. Another type of key is called the **master key**, which is used for the protection, encryption, decryption, and generation of other keys.

There are different methods to generate keys. These methods are listed as follows:

- Random, where a random number generator is used to generate a random set of bytes that can be used as a key.
- Key derivation-based, where a single key or multiple keys are derived from a password. A **key derivation function** (**KDF**) is used for this purpose, taking the password as input, and converting that into a key. Commonly used key derivation functions are **Password-Based Key Derivation Function 1** (**PBKDF1**), PBKDF2, Argon 2, and Scrypt.

KDFs are used in Ethereum wallets (keystore files) to generate an AES symmetric key that is used to encrypt the wallets. The KDF function used in Ethereum wallets is **Scrypt**. We will explain all this in detail in *Chapter 9*, *Ethereum Architecture*. More information about PBKDFs can be found at `https://tools.ietf.org/html/rfc8018`.

- Key agreement protocol, where two or more participants run a protocol that produces a key, and that key is then shared between participants. In key agreement schemes, all participants contribute equally in the effort to generate the shared secret key. The most used key agreement protocol is the Diffie-Hellman key exchange protocol.

Within the encryption schemes, there are also some random numbers that play a vital role during the operation of the encryption process. These random numbers are explained as follows:

- The **nonce** is a number that can be used only once in a cryptographic protocol. It must not be reused. Nonces can be generated from a large pool of random numbers or they can also be sequential. The most common use of nonces is to prevent replay attacks in cryptographic protocols.
- The **initial value** or **initialization vector** (**IV**) is a random number, which is basically a nonce, but it must be chosen in an unpredictable manner. This means that it cannot be sequential. IVs are used extensively in encryption algorithms to provide increased security.
- The **salt** is a cryptographically strong random value that is typically used in hash functions to provide defense against dictionary or rainbow attacks. Using dictionary attacks, hashing-based password schemes can be broken by trying hashes of millions of words from a dictionary in a brute-force manner and matching it with the hashed password. If a salt is used, then a dictionary attack becomes difficult to run because a random salt makes each password unique, and secondly, the attacker will then have to run a separate dictionary attack for random salts, which is quite unfeasible.

Now that we've introduced symmetric cryptography, we're ready to look at MACs and HMACs.

Message authentication codes

Message authentication codes (**MACs**) are sometimes called **keyed hash functions**, and they can be used to provide message integrity and authentication. More specifically, they are used to provide data origin authentication. These are symmetric cryptographic primitives that use a shared key between the sender and the receiver. MACs can be constructed using block ciphers or hash functions.

Hash-based message authentication codes

Like the hash function, **hash-based MACs** (**HMACs**) produce a fixed-length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using the MAC and the receiver verifies it using the shared key. The key is hashed with the message using either of the two methods known as **secret prefix** or **secret suffix**. With the secret prefix method, the key is concatenated with the message; that is, the key comes first, and the message comes afterward, whereas with the secret suffix method, the key comes after the message, as shown in the following equations:

$$Secret\ prefix: M = MACk(x) = h(k||x)$$

$$Secret\ suffix: M = MACk(x) = h(x||k_{)}$$

There are pros and cons to both methods. Some attacks on both schemes have occurred. There are HMAC constructions schemes that use various techniques, such as **ipad** (**inner padding**) and **opad** (**outer padding**) that have been proposed by cryptographic researchers. These are considered secure with some assumptions:

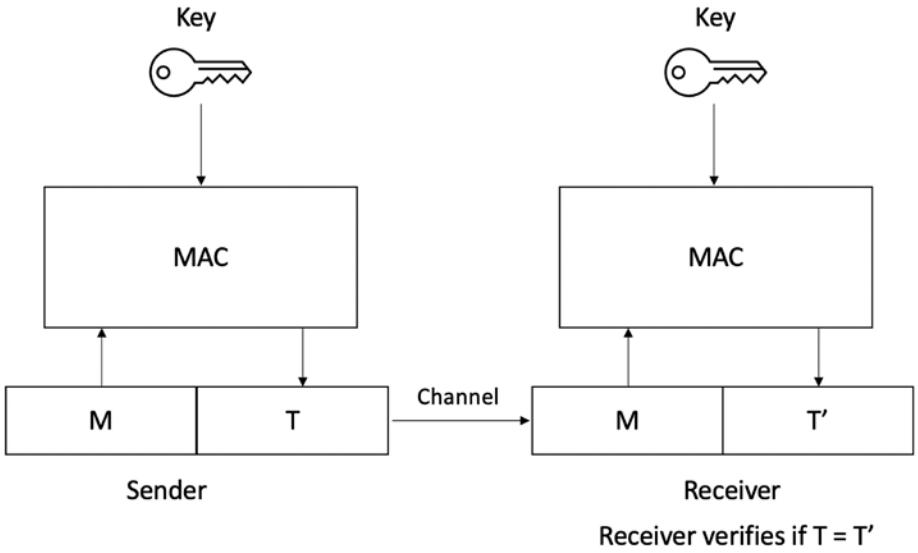


Figure 3.10: Operation of a MAC function

Secret key ciphers

There are two types of secret key ciphers or symmetric ciphers: **stream ciphers** and **block ciphers**.

Stream ciphers

Stream ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis (one bit at a time) to plaintext using a keystream.

In stream ciphers, encryption and decryption are the same functions because they are simple modulo 2 additions or **XOR** operations. The fundamental requirement in stream ciphers is the security and randomness of keystreams. Various techniques ranging from PRNGs to true hardware RNGs have been developed to generate random numbers, and it is vital that all key generators be cryptographically secure:

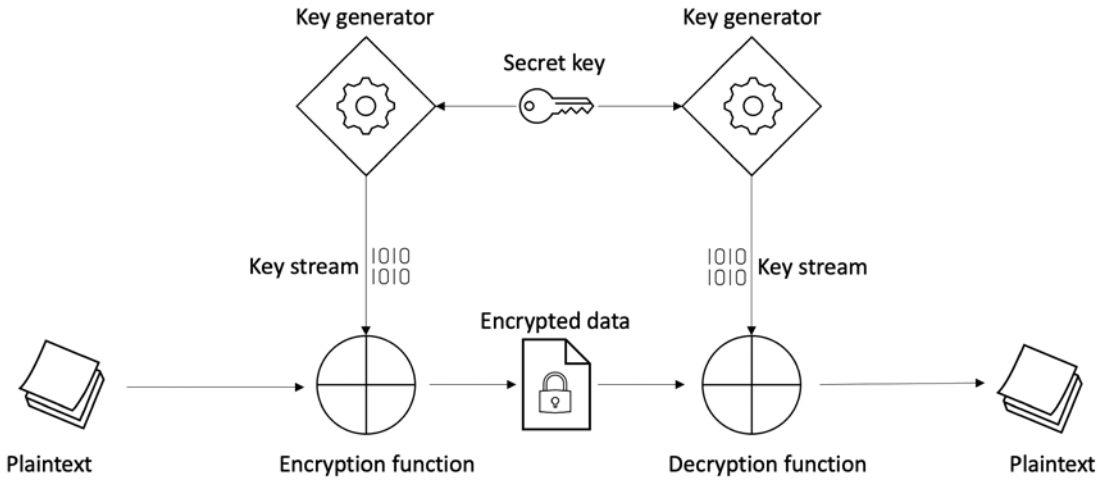


Figure 3.11: Operation of a stream cipher

There are two types of stream ciphers:

- Synchronous stream ciphers are those where the keystream is dependent only on the key.
- Asynchronous stream ciphers have a keystream that is also dependent on the encrypted data.

This concept can be visualized in *Figure 3.12* below.

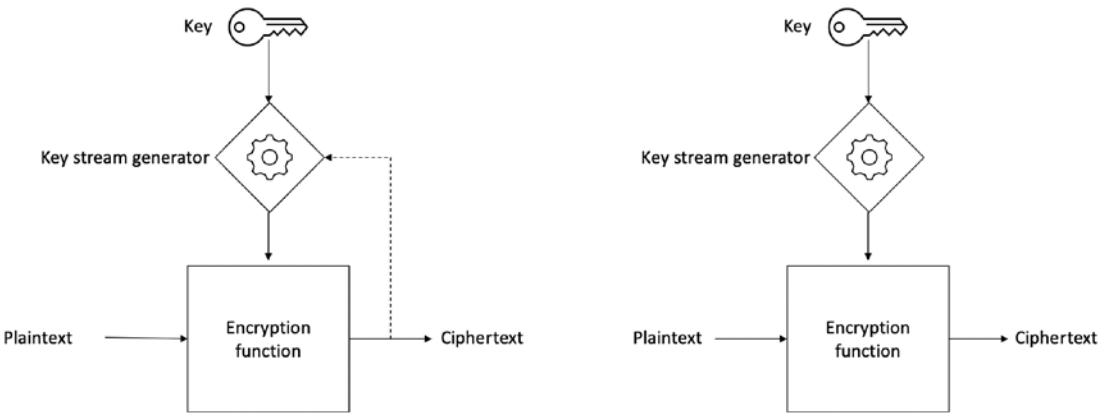


Figure 3.12: Asynchronous stream cipher (left) vs synchronous stream cipher (right)

RC4 and A5 are commonly used stream ciphers.

Block ciphers

Block ciphers are encryption algorithms that break up the text to be encrypted (plaintext) into blocks of a fixed length and apply the encryption block by block. Block ciphers are generally built using a design strategy known as a **Feistel cipher**. Recent block ciphers such as AES (Rijndael) have been built using a combination of substitution and permutation called a **substitution-permutation network** (**SPN**). **Data Encryption Standard** (**DES**) and **Advanced Encryption Standard** (**AES**) are typical examples of block ciphers.

Feistel ciphers are based on the Feistel network, which is a structure developed by Horst Feistel. This structure is based on the idea of combining multiple rounds of repeated operations to achieve desirable cryptographic properties known as **confusion** and **diffusion**. Feistel networks operate by dividing data into two blocks (left and right) and processing these blocks via keyed **round functions** in iterations to provide sufficient pseudorandom permutations.

Confusion adds complexity to the relationship between the encrypted text and plaintext. This is achieved by substitution. In practice, *A* in plaintext is replaced by *X* in encrypted text. In modern cryptographic algorithms, substitution is performed using lookup tables called **S-boxes**. The **diffusion** property spreads the plaintext statistically over the encrypted data. This ensures that even if a single bit is changed in the input text, it results in changing at least half (on average) of the bits in the ciphertext. Confusion is required to make finding the encryption key very difficult, even if many encrypted and decrypted data pairs are created using the same key. In practice, this is achieved by transposition or permutation.

A key advantage of using a Feistel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process to achieve decryption. DES is a prime example of Feistel-based ciphers:

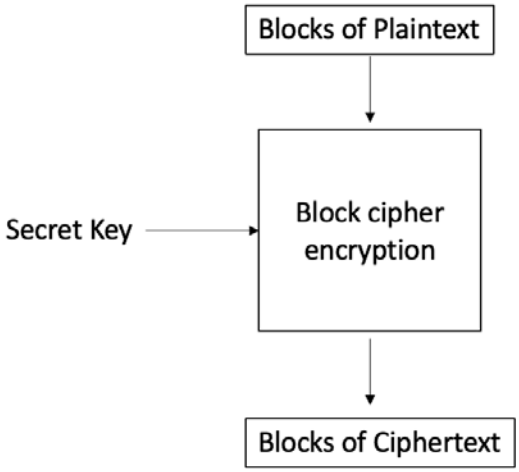


Figure 3.13: Simplified operation of a block cipher

Various modes of operation for block ciphers are used to specify the way in which an encryption function is applied to the plaintext. Some of these modes of block cipher encryption are introduced here.

Block encryption modes

In **block encryption mode**, the plaintext is divided into blocks of fixed length depending on the type of cipher used. Then the encryption function is applied to each block. The most common block encryption modes are ECB, CBC, and CTR.

Electronic code book (**ECB**) is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm to each block of plaintext, one by one. This is the most straightforward mode, but it should not be used in practice as it is insecure and can reveal information:

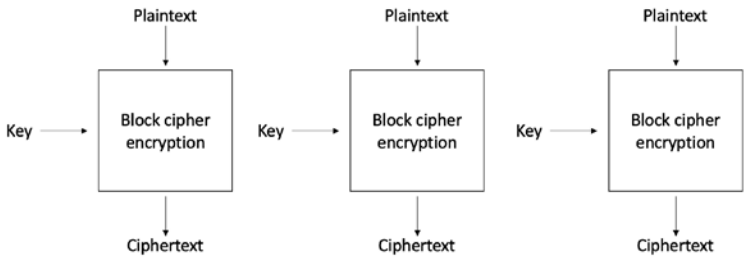


Figure 3.14: Electronic codebook mode for block ciphers

The preceding diagram shows that we have plaintext **P** provided as an input to the block cipher encryption function along with a key, and ciphertext **C** is produced as output.

In **cipher block chaining** (**CBC**) mode, each block of plaintext is XORed with the previously encrypted block. CBC mode uses the IV to encrypt the first block. It is recommended that the IV be randomly chosen:

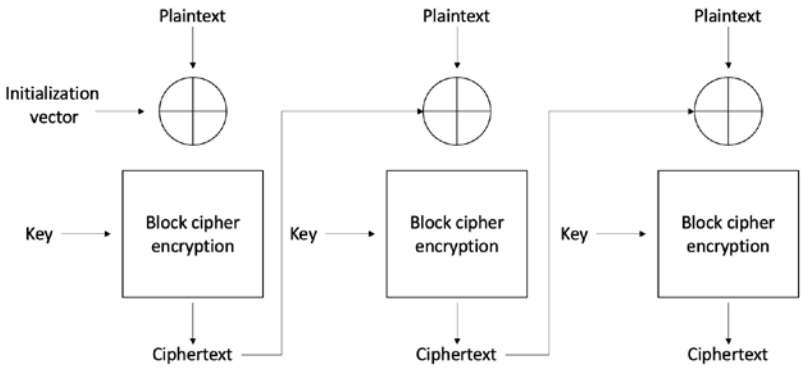


Figure 3.15: Cipher block chaining mode

The **counter (CTR) mode** effectively uses a block cipher as a stream cipher. In this case, a unique nonce is supplied that is concatenated with the counter value to produce a **keystream**:

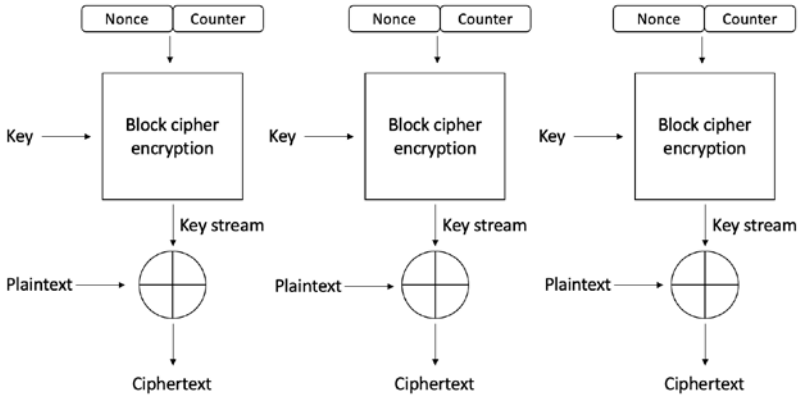


Figure 3.16: Counter mode

CTR mode, as shown in the preceding diagram, works by utilizing a **nonce** (**N**) and a **counter** (**C**) that feed into the **block cipher encryption** function. The block cipher encryption function takes the **secret key** (**KEY**) as input and produces a keystream (a stream of pseudorandom or random characters), which, when XORed with the **plaintext** (**P**), produces the **ciphertext** (**C**).

So far, we have discussed modes that are used to produce ciphertexts (encrypted data). However, there are other modes that can be used for different purposes as listed below:

- In **keystream generation mode**, the encryption function generates a keystream that is used in stream ciphers. Specifically, the keystream is usually XORed with the plaintext stream to produce encrypted text.
- In **message authentication mode**, a MAC is produced from an encryption function. A MAC is a cryptographic checksum that provides an *integrity service*. The most common method to generate a MAC using block ciphers is CBC-MAC, where a part of the last block of the chain is used as a MAC. In other words, block ciphers are used in the **cipher block chaining mode** (**CBC mode**) to generate a MAC. A MAC can be used to check if a message has been modified by an unauthorized entity. This can be achieved by encrypting the message with a key using the MAC function. The resulting message and the MAC of the message, once received by the receiver, are checked by encrypting the message received, again with the key, and comparing it with the MAC received from the sender. If they both match, then it means that the message has not been modified by some unauthorized entity, thus an integrity service is provided. If they don't match, then it means that the message has been altered by some unauthorized entity during transmission.
- Any block cipher, for example, AES in CBC mode, can be used to generate a MAC. The MAC of the message is, in fact, the output of the last round of the CBC operation. The length of the MAC output is the same as the block length of the block cipher used to generate the MAC. It should also be noted that MACs work like **digital signatures**. However, they cannot provide a non-repudiation service due to their symmetric nature.
- Hash functions are primarily used to compress a message to a fixed-length digest. In **cryptographic hash mode**, block ciphers are used as a compression function to produce a hash of plaintext.

There are also other modes, such as **cipher feedback** (**CFB**) mode, **Galois counter** (**GCM**) mode, and **output feedback** (**OFB**) mode, which are also used in various scenarios. Discussion of all these different modes is beyond the scope of this book. Interested readers may refer to any standard textbook on cryptography for further details.

We now introduce some concepts that are relevant to cryptography and are extensively used in many applications.

Advanced Encryption Standard

In this section, we will introduce the design and mechanism of a currently market-dominant block cipher known as AES.

Before discussing AES, let's review some history of DES that led to the development of the new AES standard.

Data Encryption Standard

DES was introduced by **NIST** as a standard algorithm for encryption, and it was in widespread use during the 1980s and 1990s. However, it did not prove to be very resistant to brute-force attacks, due to advances in technology and cryptography research. In July 1998, for example, the **Electronic Frontier Foundation** (**EFF**) broke DES using a special-purpose machine called the EFF DES cracker (or **Deep Crack**).

DES uses a key of only 56 bits, which raised some concerns. This problem was addressed with the introduction of **Triple DES** (**3DES**), which proposed applying a DES cipher three times to each block, thus making the key size 168 bits. This technique makes brute-force attacks almost impossible. However, other limitations, such as slow performance and a small 64-bit block size, were not desirable.

How AES works

In 2001, after an open competition, an encryption algorithm named Rijndael invented by cryptographers Joan Daemen and Vincent Rijmen was standardized as **AES** with minor modifications by NIST. So far, no attack has been found against AES that is more effective than the brute-force method. The original version of Rijndael permits different key and block sizes of 128 bits, 192 bits, and 256 bits. In the AES standard, however, only a 128-bit block size is allowed. However, key sizes of 128 bits, 192 bits, and 256 bits are permissible.

During AES algorithm processing, a 4×4 array of bytes known as the **state** is modified using multiple rounds. Full encryption requires 10 to 14 rounds, depending on the size of the key. The following table shows the key sizes and the required number of rounds:

Key size	Number of rounds required
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

Once the state is initialized with the input to the cipher, the following four operations are performed sequentially step by step to encrypt the input:

a. **AddRoundKey**: In this step, the state array is XORed with a **subkey**, which is derived from the master key.
b. **SubBytes**: This is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array.
c. **ShiftRows**: This step is used to shift each row to the left, except for the first one, in the state array in a cyclic and incremental manner.
d. **MixColumns**: Finally, all bytes are mixed in a linear fashion (linear transformation), column-wise.

This is one round of AES. In the final round (either the 10th, 12th, or 14th round, depending on the key size), stage 4 is replaced with **AddRoundKey** to ensure that the first three steps cannot be simply reversed, as shown in the following diagram:

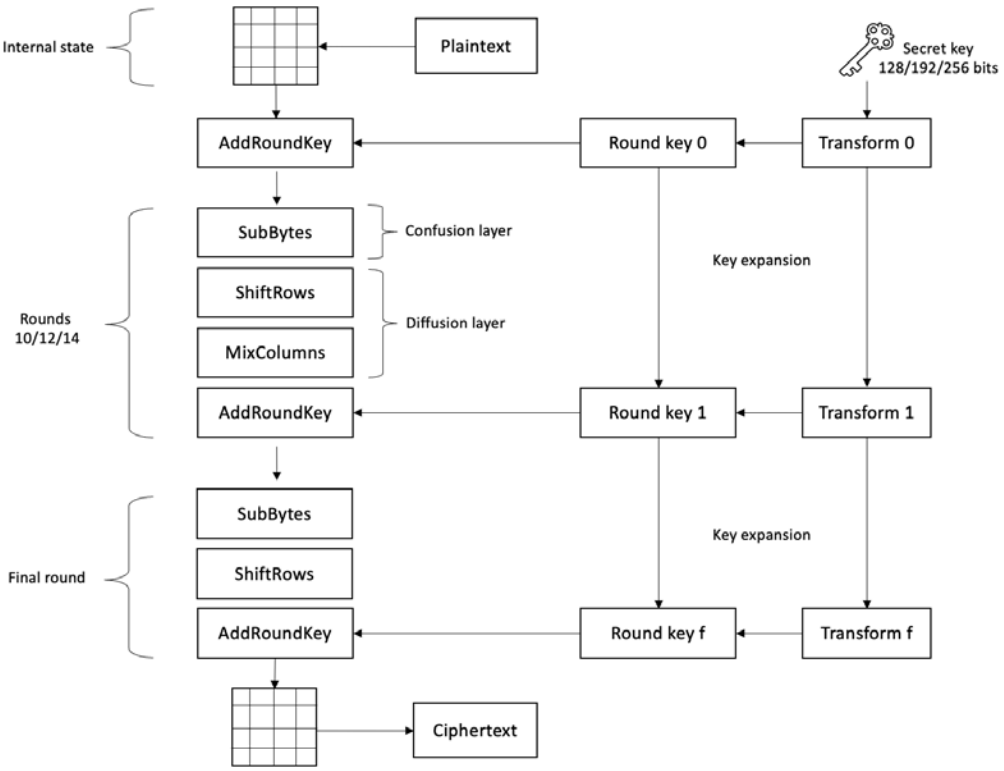


Figure 3.17: AES block diagram, showing the first round of AES encryption. In the final round, the mixing step is not performed

Various cryptocurrency wallets use AES encryption to encrypt locally stored data. Bitcoin wallets use AES-256 in CBC mode to encrypt the private keys. In Ethereum wallets, AES-128-CTR is used; that is, AES 128-bit in counter mode is used to encrypt the private key. Peers in Ethereum also use AES in counter mode (AES CTR) to encrypt their **P2P** communication.

Encrypting and decrypting using AES

We can use the OpenSSL command-line tool to perform encryption and decryption operations. An example is given here.

First, we create a plaintext file to be encrypted:

```
$ echo Datatoencrypt > message.txt
```

Now the file is created, we can run the OpenSSL tool with appropriate parameters to encrypt the file `message.txt` using 256-bit AES in CBC mode:

```
$ openssl enc -aes-256-cbc -in message.txt -out message.bin
```

It will prompt for the password:

```
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
```

Once the operation completes, it will produce a `message.bin` file containing the encrypted data from the `message.txt` file. We can view this file, which shows the encrypted contents of the `message.txt` file:

```
$ cat message.bin
```

This shows the encrypted output below:

```
Salted__?W?~?;??G+??"f??%
```

Note that `message.bin` is a binary file. Sometimes, it is desirable to encode this binary file in a text format for compatibility/interoperability reasons. A common text encoding format is `base64`. The following commands can be used to create a base64-encoded message:

```
$ openssl enc -base64 -in message.bin -out message.b64
```

Enter the command below to view the file:

```
$ cat message.b64
```

This shows the base64-encoded output below:

```
U2FsdGVkX1/tEFeZfszXiB47pOt/RyuN/CJm/x/KBBw=
```

To decrypt an AES-encrypted file, the following commands can be used. An example of `message.bin` from a previous example is used:

```
$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec
```

It will ask for the password:

```
enter aes-256-cbc decryption password:
```

Once entered, execute the following command:

```
$ cat message.dec
```

This shows the output below, the original plaintext:

```
Datatoencrypt
```

Readers may have noticed that no IV has been provided, even though it's required in all block encryption modes of operation except ECB. The reason for this is that OpenSSL automatically derives the IV from the given password. Users can specify the IV using the `-iv` switch as shown below:

```
-iv val
```

Here, `val` is IV in hex. In order to decode from base64, the following commands are used. Follow the `message.b64` file from the previous example:

```
$ openssl enc -d -base64 -in message.b64 -out message.ptx
$ cat message.ptx
```

This shows the output below:

```
Salted__?W?~??;??G+??"f??%
```

There are many types of ciphers that are supported in OpenSSL. You can explore these options based on the examples provided thus far. Further information and help can be retrieved with the following command:

```
$ openssl help
```

We will also use OpenSSL in the next chapter to demonstrate various public key cryptographic primitives and protocols.

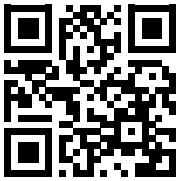
Summary

This chapter introduced symmetric key cryptography. We started with basic mathematical definitions and cryptographic primitives. After this, we introduced the concepts of stream and block ciphers along with the working modes of block ciphers. Moreover, we introduced some practical exercises using OpenSSL to complement the theoretical concepts covered.

In the next chapter, we will introduce public key cryptography, which is used extensively in blockchain technology and has very interesting properties.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`